Efficient Use of Numeric and Character Data Types

Una Smith, Duke University

Abstract

A basic understanding of data types in computer programming languages is necessary for the efficient construction of both data sets and programs in the SAS System. This paper addresses the efficient use of numeric and character data types at two levels of project development: initial planning and ongoing maintenance.

For researchers and planners, some general concepts of good data design are outlined, with an emphasis on data types; these concepts should help you to design data sets that are both clearer and easier to work with. SAS users who have not agonized over the choice of numeric or character variables may discover that there are good reasons for exercising care in this matter. Those who have may find a review of the major issues helpful as they learn how to use the powerful SAS data step more efficiently.

For the data base manager, some practical strategies for working around sub-optimal types are described. The significance of numeric variable lengths for storage requirements on various computer platforms is covered, and the use and misuse of boolean, concatenated, date, time and datetime values is discussed. Important points are illustrated with complete SAS data step programs.

1 Introduction

Experienced users of the SAS System typically have four goals in mind when thinking about good data design:

- To reduce CPU usage or improve turn-around time.
- To reduce storage requirements.
- To enhance program clarity, elegance and convenience.
- To enhance end-user quality control and data safety.

Progress toward the first two of these goals can be precisely stated in terms of the computer resources they use. Frequently, there is a tradeoff between the two; in order to write faster data steps, SAS programmers must often larger data sets. This usually occurs when character variables are replaced with numeric ones, or vice versa. The interaction may also be a positive one, with improvements on both counts; this tends to occur most often when the two remaining goals, convenience and control, are kept in mind. These two goals are very subjective; experienced users gain an intuitive sense about which techniques are effective for the kinds of data they work with. There is generally a tradeoff between these goals as well; by definition, expert users are capable of writing perfectly correct programs that most inexperienced users would not understand.

"Efficiency" can mean tuning your code for each of these, but there may be unexpected interactions between them; I make no attempt at providing the "right" answer, only suggestions for thoughtful decision-making; they may help you find the best solution for your own computing problems.

In particular, I show how some common usages of numeric and/or character types affect each strategy. When you return to your familiar data sets at the end of this conference, you may well see some of your favorite variables in a new light. Or maybe not. Either way, you will have a better feeling for the data; you will see more clearly what the data is, not just what it looks like. Also, you will be better able to choose between numeric and character variables to make your data sets and programs both more efficient and easier to understand.

---

1 SAS is a registered trademark of SAS Institute, Inc., Cary, NC, USA. The word SAS, when used as a noun in this paper, refers to the SAS programming language.
2 Data Types

Data types are abstractions that nonetheless have very powerful, almost physical roles in computer function. A data type is a contract between your software and your computer, whereby a name or number stored in an electrical circuit somewhere inside the computer's memory can be retrieved again in a form that the software recognizes.

A data type is also a contract between your software and you, specifying exactly what you may do to data of that type. The functions and procedures built into a software package also use data types to define the types of input they require and output they create. Each function and procedure specifies exactly the data type that each value must have when use as an argument to it. Likewise, the result of a function has a predefined data type. Note that, by definition, a procedure may modify the values of its arguments, but does not give a separate result. Also note that a value may be either a constant, or the contents of a variable.

2.1 Operators

The Pascal language has several types of operators for numeric values; division is performed by both div, which gives integer results, and /, which gives real results. SAS has a rich variety of unique operators, as well as a variety of functions that substitute for special operators. For example, the SAS boolean operators and, or, and not operate on numeric values.

2.2 Functions

As was mentioned briefly at the start of this section, the argument or arguments to each SAS function must be of a particular type, either numeric or character, and the result of each function is also of a particular type. The SAS functions are grouped, more or less according to these data types, as follows:

1. numeric functions, including
   (a) Arithmetic
   (b) Financial
   (c) Mathematical
   (d) Probability distributions
   (e) Quantiles
   (f) Random number generation
   (g) Sample statistics
   (h) Trigonometric and hyperbolic
2. Array functions
3. Character functions
4. Date and time functions
5. State and Zipcode functions
6. Special functions
7. System functions
8. Truncation

For example, the SAS substr function expects three arguments; the first argument must be a value of type character, such as 'SUGI 16'. The two remaining arguments must both be numeric values, such as 1 and 4. Thus the expression

\texttt{substr('SUGI 16', 1, 4)} returns the result

'SUGI', a character value. The numbers and types of arguments expected by each SAS function are explained in detail in all versions of the SAS manuals.

2.3 Values

Most computer programming languages have large sets of data types, and allow the construction of additional user-defined types. As a rule of thumb, the more specialized a language is, the fewer data types it supports. The REXX language has no data types: SAS has two. The Pascal language, in its simplest form, includes four types: char for single characters, integer for numbers such as 0, 1, 2 etc., boolean for the values True and False, and real for numbers such as 0.0, 1.33... etc.

The SAS language includes only numeric and character data types. The type of a constant value is determined by the absence or presence of quotes around the value. The type of a variable value is determined by an attrib, length, retain or input statement, or by assignment from a function or operator.

2.4 Variables

Because SAS has only two data types, numeric and character, the choice of which to use for variables becomes an issue for careful consideration. (Relatively little thought is needed to select an appropriate data type when there are dozens of types to choose from!) SAS has several unusual conventions which superficially approximate the function of additional data types. These are the numeric SAS date, time, and datetime values and their associated SAS informats, formats and functions, and the bit mask convention along with the byte and rank functions. These conventions must be understood in detail if they are to be
used efficiently and safely. In particular, it is important to know how the number of bytes used to store a numeric variable affects the accuracy of values under these conventions. In particular, it is important to know how the number of bytes used to store a numeric variable affects the accuracy of values under these conventions. 

2.5 Informats and Formats

The utility of both data types (but particularly the numeric type) is enhanced by the use of informats and formats, and the format procedure, which enables you to create your own SAS formats. Note that user-defined informats are not allowed, for reasons that go beyond the scope of this paper. SAS informats and formats, respectively, affect interpretation of data values only on input and output; it may be helpful to think of them as filters or templates which allow data to pass through in only one direction. These and other conventions are discussed at greater length in the following sections.

3 Numeric Conventions

To represent a fractional or floating point value such as 0.333333... to the fullest precision available, a variable with a length of 8 bytes must be used. The choice of length for numeric variables depends in part on whether the values stored in that variable are of type integer, or fixed or floating point (both are of type real).

To create what passes for an integer in the SAS language, use the int, trunc, and round functions. To create a fixed point (or decimal) number, use the round function:

\[
\begin{align*}
1/3 & \rightarrow 0.333333... \\
\text{round}(1/3, 0.01) & \rightarrow 0.330000... \\
\text{round}(1/3, 1) & \rightarrow 0
\end{align*}
\]

3.1 Fixed Point

Generally, fixed point variables can be treated much like integers in the calculation of necessary lengths, if you pretend momentarily that there is no decimal point. For example, a variable whose values range between 0.0 and 9.9 by 0.1 increments is similar to a variable whose values are integers ranging between 0 and 99. For SAS version 6.04 and earlier, compare this range of 'integer' values to the values listed in the tables at the end of the section on the length statement in the SAS reference manuals. You must use a version of the manual appropriate for your computer. Also, the length you choose may be too small on another computer; this is true whenever you use less than 8 bytes to store any numeric variables in SAS.

Use extreme caution; read the various discussions on this topic in the SAS documentation and talk to a professional SAS programmer, and proceed with caution, or play it safe and use a length of 8 bytes for all variables whose values may have decimal parts.

The following is another quick and dirty, but definitive test of the least number of bytes needed to store all values of a variable correctly:

1. Use a procedure such as univariate to find the minimum and maximum values stored in the variable, if necessary.
2. Figure out how many bytes you probably need, using the technique described above, or use the least number that your SAS implementation will allow. On most computers, at least 3 bytes are needed to store any numeric variable, but on a few machines 2 bytes may be enough. If you don't want to bother with this step, use 3.

3. Run this program:

```sas
%let least = 3;
%let minimum = 0;
%let maximum = 99.99;
%let interval = 0.01;
%do bytes = %let least %to %let 8;
  data temp;
    length i %byte;
    do i = %let minimum to %let maximum by %let interval;
      output;
    end;
  run;
  proc sort nodupkeys;
    by i;
  run;
  %put 'Length of i was %byte...';
%end;
```

4. Inspect the output log. If any observations were dropped during the SORT procedure, then the number of bytes used was too small.

Another way to store fixed point data using the fewest bytes possible is to adjust the units of measure, so that the data can be stored as integer values instead. Do this only if the data are still intelligible in the new units. For example, it may make sense to express tree trunk diameters in (integer) millimeters instead of the more usual tenths of centimeters (i.e., 19 mm instead of 1.9 cm). The effort of writing down and keying in the decimal point would be saved, and the values could be displayed in centimeters at any time, simply by defining a format using the picture

331
statement of the format procedure. Unfortunately, this strategy may not work if you use units that are not metric; how would you express 10.2 inches without the decimal place?

3.2 Dates, Times and Datetimes

SAS has a special convention for storing dates as numeric values representing the number of days since January 1, 1960. A similar convention is available for storing datetime values as the number of seconds since midnight, January 1, 1960. The SAS Time value is simply the number of seconds since midnight. The SAS date convention is extremely useful for storing dates, but it has several drawbacks; the stored dates appear to be random integer values, and are not recognizable without the use of a SAS date format. Also, because these dates are actually stored as numeric values, the SAS compiler will not complain about inappropriate operations (such as multiplication) using date values.

Because they are easier to read, many SAS users prefer to use Julian date values. Unfortunately, there is a problem with this method as well: Julian dates are another example of concatenated values. A Julian date (e.g., 91051, the 51st day of '91) can be stored conveniently as a numeric value, but it is actually two values: year and day within year. Julian dates can be compared to one another safely, since they sort properly. However, they can not be used reliably to calculate intervals, nor can intervals reliably be added to or subtracted from Julian dates. Despite the drawbacks that result from not having a true data type for date values, these two formats are both powerful and convenient, when used together with the many informats, formats and functions that SAS provides to help you:

- Compare a date to other dates.
- Subtract one date from another to calculate an interval of time.
- Add or subtract integer values (days) from a date to calculate a new date.
- Use date functions, add or subtract months and years from a date.
- Use date informats and formats, read and write dates in virtually any style.
- Store dates in numeric variables as either Julian or SAS date values.
- Use The datejul and juldato functions, convert SAS date values to Julian values and from Julian values to SAS date values.
- Remember: operations other than comparisons should be performed only on SAS date values.

3.3 Boolean

Some programming languages include a true boolean type; SAS has a convention which approximates the function of a boolean data type. In SAS, an expression testing that true, a numeric variable, does not contain the value 0 would be written if true. The SAS compiler expands the expression

if true
into
if true ne 0,
not
if true eq 1

before interpreting it. The numeric value 0 is equivalent to the boolean value False; all other numeric values are equivalent to the boolean value True.

4 Numeric Concatenation

Character values are frequently concatenated. For example, variables such as city, state and zip may be combined to form a convenient new variable for use in a mailing list. There is a large set of functions for handling character values, including the very useful substr function for subsetting a concatenated character value. The statement

exchange = substr(phone, 4, 3);

is certainly concise, but is it efficient? Because substr expects a character argument and returns a character result, but both Phone and Exchange have already been declared numeric, SAS must perform two implicit type conversions. This data step took 417.3 CPU seconds to complete, and without the length statement, the variable Exchange would have been of type character. When the statement was rewritten as

exchange = substr(put(phone, 10.), 4, 3);

the data step ran 37% faster, finishing in 262.8 CPU seconds. Are these type conversions really necessary? Because substr expects a character argument and returns a character result, but both Phone and Exchange have already been declared numeric, SAS must perform two implicit type conversions. This data step took 417.3 CPU seconds to complete, and without the length statement, the variable Exchange would have been of type character. When the statement was rewritten once more to

exchange = int(mod(phone, 1e7) / 1e4);

the data step ran 68.1% faster, using only 133.2 CPU seconds. All three versions of the data step used the same amount of memory.
The lesson in this example is clear: Avoid unnecessary type conversions, but when type conversions are necessary, make them explicit. Most SAS programmers know that type conversions are expensive, but only a fraction of them realize that type conversions that are not requested through the input or put functions cost even more.

5 Extra Credit

The best way to learn how numeric and character data types affect program execution is to practice using them. Included below are two example situations in which the same algorithm is implemented using either numeric or character types. CPU times from large test runs are provided: the task of calculating disk data storage requirements is left to you.

5.1 Bit Masks

SAS has several functions designed for manipulating the individual bits of a value (which can be either numeric or character, but is generally character). Before attempting to use bit masks to perform bit testing, read the SAS documentation carefully. If you use bit masks to store and retrieve data, you must understand the significance of the ASCII and EBCDIC standards for representing data; they use different bit patterns to represent the same value.

If you transfer a data file between an ASCII computer such as a PC and an EBCDIC computer such as an IBM mainframe (in either direction), the bit patterns in the file will be changed so that the values are kept the same.

If you store data either an ASCII bit pattern or an EBCDIC bit pattern, you must take special precautions to preserve the bit pattern whenever moving the data to another computer platform. Also, beware that the bit mask convention does not work in the same way for the two SAS data types, and that the numeric value 1991 is stored using a bit pattern that is completely different from the pattern used to store the character value ‘1991’.

The use of bit masks is shown best with a sample SAS program. Imagine that you have collected data on the simple binomial responses of three subjects to a series of 8 tests, and that the results are conveniently recorded as a string of 1’s and 0’s in a computer file named testbits.dat:

```
10100000
01000100
00000000
```

Here is a SAS/PC program, testbits.sas:

```sas
filename bits 'testbits.dat';
data example (drop=i);
  length number 3;
  infile bits;
  ** variable in order right to left ;
  input (x8 x7 x6 x5 x4 x3 x2 x1) ($1.);
  ** store data as bits ;
  number = 0;
  array bit[8] x1-x8;
  do i = 1 to 8;
    ** expensive exponential in loop ;
    if bit[i] = '1'
      then number + 2**((i - 1));
    end;
  char = byte(number):
  ** finally, inspect with bit masks ;
  if char = '1....1....' then x3andx7 = 1;
  if char = '00000000' then none = 1;
  if char = '0.......b' then not8 = 1;
  run;
```

The value of the variable x3and7 will be 1 for the second subject and missing for the other two. The variable none will equal 1 for the second subject only, while the variable not8 will have the value 1 for both the second and the third subject.

5.2 Check Digits

If you are responsible for the design of large or expensive projects in which data-entry personnel key in multiple data sets whose records are eventually matched using a unique record key, you should probably be using a check digit scheme to ensure the accuracy of those keys. Check digits are useful only if you are able to assign the unique key, including a check digit, before the data is collected.

With careful management, check digits can be used with numbers such as SSN’s. However, in situations where the use of check digits is practical, it is often practical also to generate unique serial numbers using a simple iterative routine. For example, credit card companies and banks typically assign account numbers whose last digit is a check digit. Whenever you call a credit card company to enquire about your account, the agent’s computer verifies that the last digit matches the rest, according to a fairly simple algorithm, before attempting to locate the account number in its data base. Thus the correctness of the number is checked at the point of data entry, where any error can be instantly repaired. Some mistakes will be missed, but the most common errors (transpositions and simple substitutions) are caught immediately and effortlessly.
The check digit algorithm is as follows:

1. Some definitions are in order: Note that the words digit, numeral, and number are not interchangeable. For example, the first digit of the number 23, counting from the left, is the numeral 2; the digit and the number are both odd, but the numeral is even.

2. Consider a unique key or serial number to be a fixed-length string of digits.

3. Working from left to right, double each odd digit, subtracting 9 if the result is greater than 9. Note that this is the same as adding together the two digits of the result. For example, if the digit is 9, then twice 9 is 18, 18 minus 9 equals 9, which equals 1 plus 8:

\[
9 \times 2 = 18 \\
18 - 9 = 9 = 1 + 8
\]

4. Sum the results of the odd digit calculations and the values of all the even digits:

\[
234768891 - \\
(2 + 2) + 3 + (4 + 4) + 7 + (6 + 6 - 9) + \\
8 + (8 + 8 - 9) + 9 + (1 + 1)
\]

= 51

5. Divide the sum by 10 and take the remainder. Subtract the remainder from 10, and again divide the result by 10 and take the remainder:

\[
51/10 = 5, \text{remainder} 1 \\
10 - 1 = 9 \\
9/10 = 0, \text{remainder} 9
\]

This new remainder is the check digit:

\[
234768891 \rightarrow 2347688919
\]

To verify a number whose last digit is a check digit, strip off the check digit and use the remaining string of digits to calculate a new check digit. Compare the two digits; if they do not match, the number has been miskeyed. Use some actual MasterCard or Visa credit card numbers to verify the correctness of your SAS program, and include some 'mistakes' to see what sorts of errors the algorithm does not catch. (And be sure not to leave the credit card numbers lying around when you are through.)

Several versions of a SAS program implementing this algorithm are given below. The data _null_ step reads in a data set containing serial, a 9-digit character value. The program assumes that all values are right-adjusted and padded with 0's on the left. A check digit is calculated by accumulating the value of each digit of serial in the numeric variable check. Note that use of a numeric variable is required by this algorithm, because of the addition step.

The accumulation is done in two parts, first for the odd digits, then for the even ones. The odd digits (working from left to right) are transformed via a user-defined character format. Remember that formats work only with the put function and put statement, that the type of the format determines the expected type of the argument to the put function, and that the put function returns only character values. An input function and a numeric informat are needed to convert the transformed character value to a numeric value.

Once the calculation of check is complete, the value is converted back to a character value and stored in Valid. (It is probably a good idea to declare the check digit to be of the same type as the value it checks.) To modify this program for your own use, you need only change the %LET statements.

```
%let library = sugi;
%let dataset = nocheck;
%let idnum = serial;
%let idlen = 9;
%let idchk = valid;

libname &library 'a';

data &library..&dataset(drop=i);
do i=1 to &idlen by 2; /* odd digits */
    check + input(translate(substr(idnum, i, 1), '0246813579', '0123456789'), 1.);
end;

checksum = mod(10 - mod(check, 10), 10);
%let idchk = put(checksum, 1.);
```
The data _null_ step took 254.0 CPU seconds to process 100,000 observations. By using the format procedure and put function in place of translate in the first do loop, like this:

```
proc format;
value $odd
'0' = '0'
'1' = '1'
'2' = '2'
'3' = '3'
'4' = '4'
'5' = '5'
'6' = '6'
'7' = '7'
'8' = '8'
'9' = '9';
and
```

```
check + input(put(substr(idnum, i, 1.), $odd.), 1.);
```

Instead of

```
check + input(translate(substr(idnum, i, 1.), '0246813579', '0123456789'), 1.);
```

The modified data step ran in 102.8 seconds, or 14.5% of the original time. Now, since the addition step can not be avoided, why not convert each digit to a numeric value at the start, before any calculations? A new numeric variable, Digit, is needed:

```
** convert each digit to numeric value;
```

```
data _null_;  
set &library.. &dataset;  
check = 0;  
digit = 0;  
do i=1 to tidlen by 2; ** odd digits ;  
digit = 2 *  
      input(substr(idnum, i, 1.), 1.);  
      if digit > 9  
      then digit = digit - 9;  
      check + digit;  
      end;  
  do i=2 to tidlen by 2; ** even digits ;  
  check +  
  input(substr(idnum, i, 1.), 1.);  
  end;  
  check = mod(10 - mod(check, 10), 10);  
&kidchk = put(check, 1.);  
run;
```

This version took 77.9 CPU seconds (11.0% as long as the original) to complete. Why not convert serial to a numeric variable at the start? The algorithm might then be written:

```
** convert serial to numeric value ;
```

```
data _null_;  
set &library.. &dataset;  
check = 0;  
digit = 0;  
number = input(idnum, &idlen..);  
i = &idlen;  
do while(number > 0);  
  digit = mod(number, 10);  
  number = int(number / 10);  
  if mod(i, 2) = 1 then do;  
    digit = 2 * digit;  
    if digit > 9  
    then digit = digit - 9;  
    end;  
  i = i - 1;  
  check + digit;  
  end;  
  check = mod(10 - mod(check, 10), 10);  
&kidchk = put(check, 1.);  
run;
```

ran in 75.8 CPU seconds, only 10.7% as long as the first version using translate, and only 73.8% as long as the version using put (. .. , $odd.). Had serial been a numeric variable, no doubt the program would have run somewhat faster than this last example, since two type conversions per observation would be eliminated. However, the savings probably would not be very substantial, since the numeric version performs, in effect, ten extra divisions and multiplications for every observation.

6 Points of View

Experienced SAS programmers tend to hold one of two extreme points of view on the best method of assigning numeric or character data types to variables.

6.1 Numeric

Declare variables numeric unless the values contain non-numeral characters. (Use numerals whenever possible.)

New users generally employ this strategy also, encouraged by the syntax of the `input` statement, which assumes all data will be numeric unless otherwise indicated. Many experienced users find typing the
required quotes around character constants in their code to be a nuisance. Some long-time users will recall the difficulties that could result from using keyboards with ' and ' keys, and monitors that displayed both as '; an accidental, invisible ' can be very hard to locate. Numeric constants do not require quotes, and the special conventions, informats, and formats (including picture formats) for numeric values make producing elaborate output extremely easy for skilled programmers. This strategy emphasizes programmer convenience.

6.2 Character

Declare character variables unless the values are quantities.

This strategy is popular with managers of large, expensive and/or sensitive projects, especially when fairly inexperienced users have access to the data. The mean value of a variable such as ssn or phone may be extremely expensive to calculate, and entirely inappropriate; assigning character types to non-quantitative data helps to prevent such blunders. If the numeric type is assigned only to quantities, the special statement arguments _numeric and _character can be used very effectively in most SAS procedures, as well as the data step. This strategy is an excellent defensive programming technique.

7 Summary

- Assign types to variables according to how you expect to use the data, and according to which SAS data type best reflects the data's true type.
- Always perform explicit type conversions; they are much faster and safer than implicit conversions, and use no more memory.
- Use notes in the SAS log about type conversions to identify costly bugs in your program.
- When both argument(s) and result are numeric, try to perform numeric calculations rather than converting to character and back again.
- Understand how the input and put functions work, and use them to correct for badly-typed values.
- Avoid confusing data types with SAS formats and informats.
- Assign data types to variables in a consistent style.
- Ask another SAS programmer how he or she would implement your algorithm, or write two different versions of a SAS program that performs your algorithm, and compare the two.

8 Additional Reading


9 Acknowledgements

The inspiration for this paper was provided by the SAS-L electronic mail discussion group, an informal and extremely supportive SAS users group to which anyone may contribute, provided they have a computer userid with access to any of the worldwide computer networks. Roger Lustig contributed the check digit algorithm.